Processes, Threads, Concurrency

A process is a sequential flow of control that is able to cooperate with other processes.

Categories of processes

- Heavy-weight ↔ light-weight
  - Address space: own ↔ common (but own stack)
  - Context: large (file descriptors, I/O buffers etc.) ↔ small (the actual registers only)
  - Context-switch: expensive ↔ cheap
  - Examples: Unix or Windows/NT processes ↔ Threads ("thread of control")

- Parallel ↔ Quasi parallel (concurrent) processes
  - Processor: own ↔ common
  - Execution: true parallel ↔ virtually parallel

Process Cooperation

- Process synchronization

- Process communication

Way of Cooperation

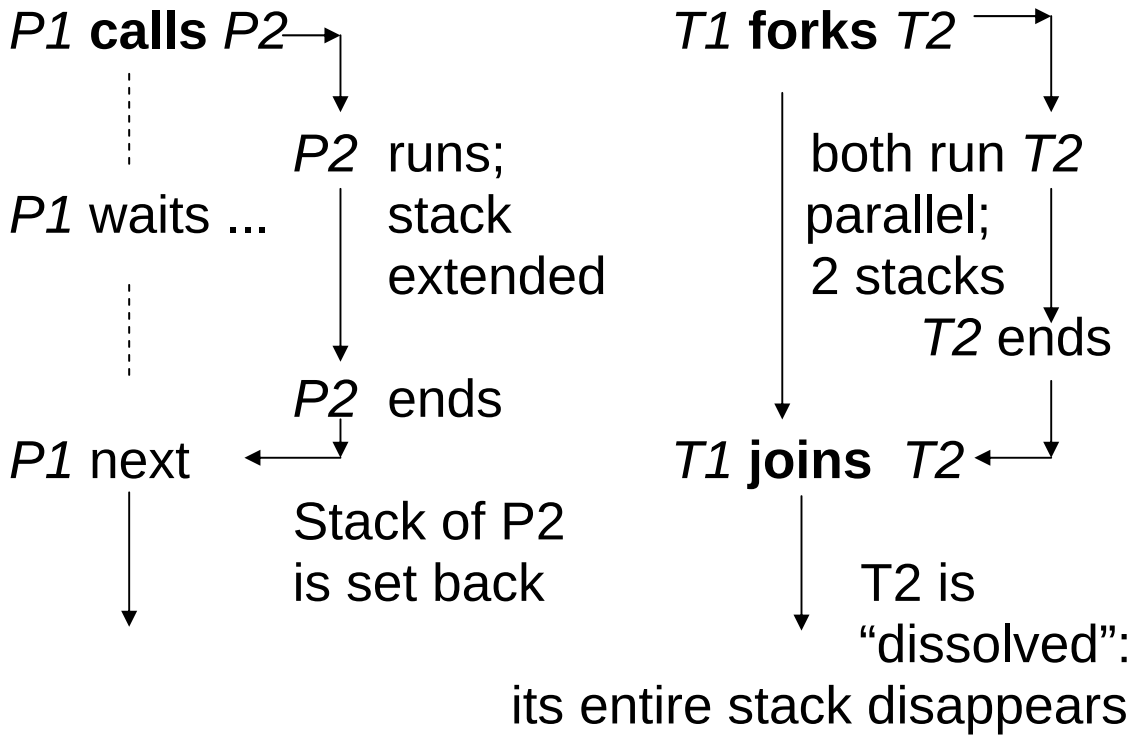
- Common memory (in threads)

- Communication channels, messages (in processes)

Fork and Join

Threads are procedures started in a special way: by fork

Let be P1, P2, T1 and T2 procedures



Concurrent, Distributed and Parallel Programming

Common features

- The program consists of more than one thread of control
- No explicit assumption about time (as opposed to *real-time* programming)

Concurrent Programming

- Main goal: Inherent parallelism (concurrency)
- Based on (generally quasi-parallel) threads, normally uses common memory, often well supported by general-purpose programming languages (e.g. Conc. Pascal, Modula-3, Java)

Parallel Programming

- Main goal: Speed-up and efficiency ($S_n = T_1/T_n$, $E_n = S_n/n$)
- Mostly based on threads, uses common or distributed memory, hardware support, often supported by special languages (e.g. High-Speed Fortran, Vienna Fortran)

Distributed Programming

- Main goal: Physical distribution (for several reasons)
- Sometimes supported by special languages (Orca), uses distributed memory, socket or RPC-based communication

Safety and Liveness

Safety: The property that nothing bad ever happens

Liveness: The property that anything ever happens at all

Sequential Programs

- Safety: partial correctness (show that the result is correct)
- Liveness: total correctness (show that the program terminates)

Parallel/Concurrent/Distributed Programs

- more difficult to ensure and prove safety and liveness
- impossible to check them fully automatically

Threads in Java

Language-level support for thread creation, control, mutual exclusion and conditional synchronization (very similar to Modula-3). The support is provided by *java.lang.Thread*.

Thread creation and control

- A thread is created by creating an instance of either
  - A class that implements the *Runnable* interface or
  - A subclass of the class *Thread*
- In both cases the *run* method must be overridden by the required thread code
- The call of *start* causes the *run* method to be started not as a procedure but as a concurrent thread (with own stack)
- *join* suspends the caller until the target thread completes
- *isAlive* returns true if the thread has been started but has not terminated (it returns true for suspended or waiting)
- *sleep* causes a thread to sleep for a given time (millisecs)
- *yield* gives a scheduling chance for other threads ≡ sleep(0)
- *stop* halts a thread, retaining all locks: DO NOT USE
- *suspend* temporarily halts a thread (retains all locks), it can be continued by *resume* – unsafe and obsolete: DO NOT USE them, use rather *wait/notify*
- *interrupt* causes a *sleep*, *wait* or *join* to abort with *InterruptedException*
  - should not be used for normal communication
  - I/O cannot be interrupted in some implementations or causes different exceptions: DO NOT USE often
  - *interrupted():* checks dynamically
  - *isInterrupted()*: checks statically

    interrupted ≡ Thread.currentThread().isInterrupoted

Mutual Exclusion

- The *synchronized* keyword can mark as critical section
  - an entire procedure
  - a piece of code inside a method

- If all methods are marked as synchronized: *monitor*

- The *volatile* keyword can suppress caching
  - Compiler + JVM may locally cache variables (for optimization reasons)
  - In multi-threaded programs this may lead to synchronization errors, if synchronized and unsynchronized accesses are mixed and/or we ues
    - Native methods
    - Busy loops
  - This kind of optimization can be suppressed by *synchronized* and *volatile*

Conditional Synchronization

- Every object provides the methods *wait*, *notify* and *notifyAll* (defined in *java.lang.Object*)

- Wait
  - suspends the thread
  - releases the lock on the object it waits for
  - all other locks are retained

- Waiting threads can be awaken by *notify* and *notifyAll*
  - Notification reacquires the lock
  - *Notify* wakes one thread
  - *notifyAll* wakes all threads waiting on the corresponding object
  - Signaling is explicit and delayed:
    Additional state-check is necessary
  - Usual pattern
    while (condition) try {wait();} catch (…) {};
  - Never use
    ~~if (condition) try {wait();} catch (…) {};~~

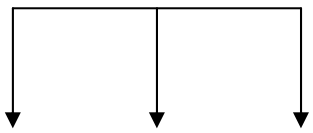
List of examples for thread control

/* Thread als Subklasse */

```
class MyThread extends Thread { /*Subklasse von Thread*/

  public void run() {                          /*Ueberschreibt Threads run*/
    System.out.println("Hello, this is " + getName());
        /*getName ist eine standard Methode der Thread API*/
  } // run

} // MyThread


public class ExtendedThread {

  static public void main(String args[]) {
    MyThread a, b;              /*a und b sind Instanzen von MyThread*/

    a = new MyThread();               /*Thread-Objekt a wird erzeugt*/
    b = new MyThread();               /*Thread-Objekt b wird erzeugt*/
    a.start();                  /*Die run-Methode von a wird geforkt*/
    b.start();                  /*Die run-Methode von b wird geforkt*/
    /*3 Threads: a, b, und main laufen nun quasi-parallel*/

    System.out.println("This is " + Thread.currentThread().getName());
    /*currentThread ist eine statische Methode der Thread Klasse*/
  } // main

} // ExtendedThread
```

```
Hello, this is Thread-0
This is main
Hello, this is Thread-1
```

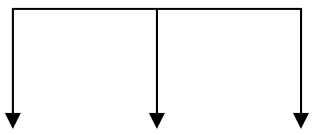
main Thread-0 Thread-1



/* Threads als Implementierung von Runnable */

```
class MyThread implements Runnable{
        /* MyThread ist keine Subklasse von Thread:
                es implementiert Runnable, das nur run enthält*/
  public void run() {                    /*Ueberschreibt Threads run*/
    System.out.println("Hello, this is " +
                    Thread.currentThread().getName());
     /* getName ist jetzt unbekannt: Runnable enthält nur run */
     /* currentThread ist eine statische Methode der Thread Klasse*/
  } // run
} // MyThread


public class RunnableThread {
  static public void main(String s[]) {
    MyThread work2do;
    Thread a, b;

    work2do = new MyThread();      /*Neue Instanz von MyThread*/
    a = new Thread(work2do);
        /*Thread Konstruktor erhält MyThread-Objekt als Parameter*/
    b = new Thread(work2do);
    a.start();                     /*Die run-Methode von a wird geforkt*/
    b.start();                     /*Die run-Methode von b wird geforkt*/

    System.out.println("This is " + Thread.currentThread().getName());
}  /*3 Threads: a, b, und main laufen quasi-parallel*/  }
```

main Thread-0 Thread-1



```
Hello, this is Thread-0
This is main
Hello, this is Thread-1
```

```
/* Fork und Join */
class MyThread extends Thread {

  public void run() {
    System.out.println( getName() + " runs" );
    for (int i = 0; i < 5; i++) {            // print thread.id 5 times
      try { sleep(500); } catch (InterruptedException e) {}
      System.out.println( "Hello,  this is " + getName() );
    } // for
    System.out.println( getName() + " finished" );
  } // run
} // MyThread

class MyThread2 extends Thread {
  private Thread wait4me;

  MyThread2(Thread target) {       // We pass the thread
    super();
    wait4me = target;                  // we want to wait for
  } // MyThread2

  public void run() {                      // wait for the other thread to join
    /* Thread may do anythig not needing the result of other */
    System.out.println( getName() + " waits on " +
                              wait4me.getName());
    try { wait4me.join(); }            // Awaits the other thread
    catch (InterruptedException e) {}
    System.out.println(wait4me.getName() + " dissolved" );
}  } // run, MyThread2

public class JoinTest {
  static public void main(String args[]) {
    MyThread a;
    MyThread2 b;
    a = new MyThread();
    b = new MyThread2( a );
    b.start();       // forks b
    a.start();       // forks a
  } // main
} // JoinTest
```

```
Thread-1 waits on Thread-0
Thread-0 runs
Hello,  this is Thread-0
Hello,  this is Thread-0
Hello,  this is Thread-0
Hello,  this is Thread-0
Hello,  this is Thread-0
Thread-0 finished
Thread-0 dissolved
```

// Anhalten mit Hilfe einer Booleschen Variable

```
class MyStoppingThread extends Thread {
  private volatile boolean isRunning = true;
// volatile garantiert, dass der Wert aktuell ist (kein Cache)

  public void stopp() {      // Ersetzt standard stop-Methode
    isRunning = false;       // Signalisiert Ende
  } // stopp

  public void run() {
    while ( isRunning ) {// Endet, wenn isRunning false wird
      System.out.println("Hello, this is " + getName());
      try { sleep(500); } catch (InterruptedException e) {}
      } // while
   } // run
} // MyStoppingThread

public class StopThread1 {
  public static void main(String[] args) {
    MyStoppingThread a;

    a = new MyStoppingThread();
    System.out.println("Start...");
    a.start();                 // Thread wird geforkt
    try {Thread.sleep(2000); }
              catch (InterruptedException e) {}

    System.out.println("Stop...");
    a.stopp();                 // Thread wird "sanft" angehalten
    try { Thread.sleep(1000); }
              catch (InterruptedException e) {}
    System.out.println("Finished...");
    } // main
 } // StopThread1
```

```
Start...
Hello, this is Thread-0
.
.
.
Hello, this is Thread-0
Stop...
Finished...
```

Safety

- Similar to type safety
  - It does not guarantee correctness but excludes errors due to incorrect thread cooperation

- Read/Write conflicts
  - Illegal transient state values can be read by a client

- Write/Write conflicts
  - Inconsistent assignment to variables by concurrently executing threads

- It is more difficult to prove and check

- Main purpose: ensuring consistent states for all objects, even for those which are used concurrently

  - *Immutability*: Avoiding state changes

  - *Synchronization*: Protect changes by dynamically ensuring exclusive access

  - *Containment*: Protect changes by structurally ensuring exclusive access

Stateless and side-effect free Methods

- Stateless methods are similar to immutable objects
  - they work only on local data (maybe on a local copy of the parameters), therefore need not be synchronized
  - Caution: No complex value parameters in Java, a local copy must be made manually!
- Examples

  - Stateless adder

```
class StatelessAdder {
  public int add(int a, int b) { return a + b; }
}
```

  - A sorter class

```
public class SortingThread extends Thread  {
        int [] p; boolean asc;        // asc is true if ascending order
        SortingThread(int [] input_arr, boolean ascending) {
                super(); p = input_arr; asc = ascending;
        } // SortingThread

        public void run()  {
                for (int i = 1; i < 1000; i++) {
                   int [] q = Sorter.Sort(p, asc);
                   Sorter.Out(q, "Output " + getName());
                   p = q;
                   try {sleep(300);} catch(InterruptedException e){}
                } // for
        } // run
} // SortingThread
```

```
public class ThreadedSort  {

        static void main(String [] args) {
                int []input_array = Sorter.Init(8);
                        // Initializes 1 array with random numbers
                Sorter.Out(input_array , "Input ");
                SortingThread         // T0 ascending, T1 descending
                        t1 = new SortingThread (input_array , true),
                        t2 = new SortingThread (input_array , false);
                t1.start();
                t2.start();
        }  // main

} // ThreadedSort
```

```
Input
61 66 19 -6 49 -58 15 22
Output Thread-0
-58 -6 15 19 22 49 61 66
Output Thread-1
66 61 49 22 19 15 -6 -58
Output Thread-0
-58 -6 15 19 22 49 61 66
Output Thread-1
66 61 49 22 19 15 -6 -58
Output Thread-0
-58 -6 15 19 22 49 61 66
Output Thread-1
66 61 49 22 19 15 -6 -58
...
```

```
public class Sorter { // Static class, no state, except random
    static Random r = new Random();

    static int [] Init (int length)  {
        int [] p = new int [length];
        for (int i = 0; i < length; i++)  p[i] = r.nextInt() % 100;
        return p;   } // Init

    synchronized static void Out (int [] p, String headLine) {
        System.out.println(headLine);
        for (int i = 0; i < p.length; i++)
            System.out.print(p[i] + " ");
        System.out.println(); } // Out

    static int [] Copy (int [] p)  {            // sync ?
        int [] q = new int [p.length];
        for (int i = 0; i < p.length; i++)  q[i] = p[i];
        return q;   } // Copy

    static int [] Sort (int [] q, boolean ascending) {
        int [] p = Copy(q);                 //Create local copy
        int minmax, x;
        for (int i = 0; i < p.length; i++)  {
            minmax = i;
            for (int j = i + 1; j < p.length; j++)
                if ( (ascending && (p[j] < p[minmax])) ||
                     (!ascending && (p[j] > p[minmax])) )
                    minmax = j;
            x = p[i]; p[i] = p[minmax]; p[minmax] = x;
            try {Thread.sleep(0); }     // Just to prvoke errors
            catch (InterruptedException e) {}
        } // for
        return p;
    } // Sort

} // Sorter
```

Liveness

- Harder than safety: mostly stems from “action at a distance”

- Often in contradiction with safety: the more synchronization the less liveness

- Contention (starvation)
  - A thread in ready state never gets running, at least not enough for meaningful work (e.g. endless busy wait)

- Dormancy
  - A thread in non- ready state never gets ready (e.g. a *wait* is never balanced by a *notify* or a *suspend* is never balanced by *resume*)

- Deadlock
  - Two or more threads block each other in a cycle, waiting for resources under mutual exclusion.
  - A deadlock may occur if all of the following 4 conditions hold:
    - Some resources need mutual exclusion
    - Cyclic waiting (e.g. *a* for *b*, *b* for *c*, *c* for *a*). More precisely: the resource allocation graph contains a cycle.
    - Hold and wait. A process waits for some new resources, while holding already acquired ones
    - No preemption. Some resources cannot be preempted (e.g. printer)

- Premature termination
  - A thread terminates earlier than it should or it is killed

Deadlock - example

```
public void removeUseless(Folder file) {
    synchronized (file) {
        if file.isUseless() {
            Directory dir = file.getDirectory();
            synchronized (dir) {
                dir.remove(file);
}   }   }   }


public void updateFolders(Directory dir) {
    synchronized (dir) {
        for (Folder f = dir.first(); f != null;
                    f = dir.next(f)) {
            synchronized (f) {
                f.update();
}   }   }   }
```

1. *Thread1*: calls *updateFolders* locks Directory *dir:* Lock **L1**
2. *Thread2*: calls *removeUseless*, locks file *f*: **L2**
3. *f* is to be removed: *Thread2* tries to lock Directory *dir* (**L1**) and waits, until *Thread1* releases it
4. *Thread1*: gets in its loop file to *f* and tries to lock it (**L2**)
5. The reference graph contains a cycle: Deadlock

- Nested Locks
  - If a thread locks the same object more than once – no problem, the system notices and allows this
  - Nested locks on different objects: danger for deadlock

Splitting Synchronization

- Try to partition into independent, non-interacting subsets
- Use separate locks for the subsets
- Example *Shape*.
  - Assumption: *adjustLocation* never deals with dimension and *adjustDimensions* with location

```
public class Shape {

  protected double
    x_= 0.0, y_=0.0, width_=0.0, height_=0.0;

  public synchronized double x() { return x_; }
  public synchronized double y() { return y_; }

  public synchronized double width(){
    return width_; }
  public synchronized double height() {
    return height_; }

  public synchronized void adjustLocation() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }

  public synchronized void adjustDimensions() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }

   protected double longCalculation1() { /*...*/

} // Shape
```

```
  protected double
    x_= 0.0, y_=0.0, width_=0.0, height_=0.0;

  protected Object locationLock_ = new Object();

  protected Object dimensionLock_ = new Object();

  public double x() {
    synchronized(locationLock_) { return x_; }
  }

  public double y() {
    synchronized(locationLock_) { return y_; }
  }

  public void adjustLocation() {
    synchronized(locationLock_) {
      x_ = longCalculation1();
      y_ = longCalculation2();
    }
  }

  public double height() {
    synchronized(dimensionLock _) { return height_; }
  }

  public double width() {
    synchronized(dimensionLock _) { return width_; }
  }

  public void adjustDimension() {
    synchronized(dimensionLock_) {
      height_ = longCalculation3();
      width_ = longCalculation4();
    }
  }

 protected double longCalculation1() { /* ... */
} // LockSplitShape
```

Conditional Synchronization

- An action has often a *precondition* that will be fulfilled *later*. In such a case *conditional waiting* is necessary. When the condition is fulfilled, waiting threads must be *notified*

- Java supports monitor-like conditional waiting and explicit notification

- Notification means only that the corresponding thread(s) will get control *sometimes* in the future. Therefore, the static state must be checked again (*while* instead of *if*)

- Example: Bounded Counter. The operations *increment* and *decrement* must be postponed until the precondition is true.

State	Condition	increment	decrement
Top	value == MAX	no	yes
Middle	MIN < value < MAX	yes	yes
Bottom	value == MIN	yes	no

```
public interface BoundedCounter {
  public static final long MIN = 0;
           // minimum allowed value
  public static final long MAX = 5;
           // maximum allowed value
  public long value(); // MIN <= value() <= MAX
           // initial condition: value() == MIN
  public void inc(); // only when value() < MAX
  public void dec(); // only when value() > MIN

} // BoundedCounter
```

- Bounded Counter, Solution-1.
- At all state changes all waiting threads are notified. Causes many unnecessary notifications

```
public class BC1 implements BoundedCounter {
  protected long count_ = MIN; // initial value
  public synchronized long value() {
                                return count_; }

  public synchronized void inc()  {
    awaitIncrementable();
    setCount(count_ + 1);
  }
  public synchronized void dec()  {
    awaitDecrementable();
    setCount(count_ - 1);
  }
  protected synchronized void setCount
                                 (long newValue) {
    count_ = newValue;
    notifyAll();
    // wake up any thread depending on new value
  }
   protected synchronized void
                         awaitIncrementable() {
    while (count_ >= MAX)
      try { wait(); }
      catch(InterruptedException ex) {};
  }

  protected synchronized void
                         awaitDecrementable() {
    while (count_ <= MIN)
      try { wait(); }
      catch(InterruptedException ex) {};
  }
} // BC1
```

- Bounded Counter, Solution-2
- Generates notifications only if necessary

```
public class BC2 implements BoundedCounter {

protected long count_ = MIN;

  public synchronized long value()
                                { return count_; }

  public synchronized void inc()  {

    while (count_ == MAX)
      try { wait(); }
      catch(InterruptedException ex) {};

    if (count_++ == MIN)
      notifyAll(); // signal if was bottom
  }

  public synchronized void dec()  {
    while (count_ == MIN)
      try { wait(); }
      catch(InterruptedException ex) {};

    if (count_-- == MAX)
      notifyAll(); // signal if was top
  }

}// BC2
```

- Examle: Bounded Buffer
- Classical monitor-like solution
- Threads wanting to put an element into a full resp. to take an element from an empty buffer wait automatically.

```
public interface BoundedBuffer {

  public int    capacity();
           // invariant: 0 < capacity

  public int    count();
           // invariant: 0 <= count <= capacity

  public void   put(Object x);
           // add only when count < capacity

  public Object get();
           // remove only when count > 0

}// BoundedBuffer
```

```
public class BB1 implements BoundedBuffer {
  protected Object[]  array_;    // content
   protected int putPtr_ = 0;     // circ. index
  protected int getPtr_ = 0;    // circ. index
  protected int usedSlots_ = 0; // no. of elems
  public BB1 (int capacity)
   throws IllegalArgumentException {
    if (capacity <= 0) throw new
           IllegalArgumentException();
    array_ = new Object[capacity];
  }
  public int count(){return usedSlots_;}//Sync?
  public int capacity() {return array_.length;}
  public synchronized void put(Object x) {
    while (usedSlots_ == array_.length)
                          // wait until not full
      try { wait(); }
      catch(InterruptedException ex) {};
    array_[putPtr_] = x;
    putPtr_ = (putPtr_ + 1) % array_.length;
                          // cyclically inc
    if (usedSlots_++ == 0)
      notifyAll();
  }
  public synchronized Object get() {
    while (usedSlots_ == 0)
                          // wait until not empty
      try { wait(); }
      catch(InterruptedException ex) {};
    Object x = array_[getPtr_];
    array_[getPtr_] = null;
    getPtr_ = (getPtr_ + 1) % array_.length;
    if (usedSlots_-- == array_.length)
      notifyAll();
    return x;
  }
}// BB1
```

```
class Producer extends Thread  {
        BoundedBuffer buf = null;
        String id = getName();
        Producer (BoundedBuffer buffer)
                { buf = buffer; }
        public void run ()  {
                for (int i = 0; i < 10; i++) {
                        String text = id + ">" + i;
                        buf.put(text);
                } // for i
                buf.put("halt");
        } // run
} // Producer

class Consumer extends Thread  {
        BoundedBuffer buf = null;
        Consumer (BoundedBuffer buffer)
                { buf = buffer;}
        String text = null;
        public void run ()  {
           do {       text = (String) buf.get();
                      System.out.println(getName() + "<" + text);
                }       while (!text.equals("halt"));
        } // run
} // Consumer

public class BufUser {
        public static void main  (String arg[]) {
                BoundedBuffer buffer = new BB1 (5);             // Buffer impl.
                Producer [] producer = new Producer [3];
                Consumer consumer = new Consumer(buffer);
                for (int i = 0; i < producer.length; i++)  { // 3 Producers start
                        producer[i] = new Producer(buffer);
                        producer[i].start();
                } // for i
                consumer.start();                       // 1 Consumer starts
                try { consumer.join(); System.exit(0); } // cons. joins main
                catch (java.lang.InterruptedException e) {}
        } // main
}// BufUser
```

```
Producer-Thread-1 starts
Producer-Thread-2 starts
Producer-Thread-3 starts
Consumer-Thread-0 starts
Thread-0<Thread-1>0
Thread-0<Thread-1>1
Thread-0<Thread-1>2
Thread-0<Thread-1>3
Thread-0<Thread-1>4
Thread-0<Thread-1>5
Thread-0<Thread-2>0
Thread-0<Thread-3>0
Thread-0<Thread-1>6
Thread-0<Thread-2>1
Thread-0<Thread-3>1
Thread-0<Thread-1>7
Thread-0<Thread-2>2
Thread-0<Thread-3>2
Thread-0<Thread-1>8
Thread-0<Thread-2>3
Thread-0<Thread-3>3
Thread-0<Thread-1>9
Thread-0<Thread-2>4
Thread-0<Thread-3>4
Thread-0<halt
```

- Example: Readers/Writers

```
public abstract class RW {
  protected int activeReaders_ = 0;  // threads reading
  protected int activeWriters_ = 0;  // always zero or one
  protected int waitingReaders_ = 0; // threads waiting to read
  protected int waitingWriters_ = 0; // same for write_

  protected abstract void read_();   // implement in subclasses
  protected abstract void write_();  // implement in subclasses

  public void read()  {
    beforeRead();
    read_();
    afterRead();
  } // read
  public void write() {
    beforeWrite();
    write_();
    afterWrite();
  } // write
  protected boolean allowReader() {
    return waitingWriters_ == 0 && activeWriters_ == 0;
  } // allowReader
  protected boolean allowWriter() {
    return activeReaders_ == 0 && activeWriters_ == 0;
  } // allowWriter
  protected synchronized void beforeRead() {
    ++waitingReaders_;
    while (!allowReader())
      try { wait(); } catch (InterruptedException ex) {}
    --waitingReaders_;
    ++activeReaders_;
  } // beforeRead
  protected synchronized void afterRead()  {
    --activeReaders_;
    notifyAll();
  } // afterRead
  protected synchronized void beforeWrite() {
    ++waitingWriters_;
    while (!allowWriter())
      try { wait(); } catch (InterruptedException ex) {}
    --waitingWriters_;
    ++activeWriters_;
  } // beforeWrite
  protected synchronized void afterWrite() {
    --activeWriters_;
    notifyAll();
  } // afterWrite
} // RW
```

- Difference between wait and sleep
  - Sleep does not release the lock
  - We have to do that “manually”

```
public class Example {
   public synchronized void ProcessLoop() {
          processOne();
          try {wait(1000);} catch (Exception e) {}
          processTwo();
}  }
```

```
public class Example {
   public void ProcessLoop() {
          synchronized  (this) { processOne(); }
          try {Thread.sleep(1000);} catch (Exception e) {}
          synchronized  (this) { processTwo(); }
}  }
```

Scheduling

Scheduling controls the transitions of threads between their states. Especially, it decides, which thread makes the transition: ready → running.

The scheduler makes scheduling decisions at scheduling points:
1. Running → Waiting (e.g. I/O is started)
2. Running → Ready (e.g. Interrupt happened)
3. Waiting→ Ready (e.g. end of I/O)
4. Termination

- Nonpreemptive Scheduling
  - Scheduling only in cases 1 and 4

- Preemptive Scheduling
  - Scheduling in all four cases
  - Threads can be forced to release the CPU
  - Time-slicing
    - Every thread gets a certain time quantum
    - If the time quantum is exhausted a timer interrupt is generated and the thread is "preempted"(is forced to the transition: Running → Ready)

- Long-term, middle-term, short-term scheduling

- Fairness
  - Each thread gets its CPU share, whatever other threads do
  - No thread misses its signal eternally
  - No thread waits in the ready queue eternally

Scheduling in Java

- Priority scheduling
  - A priority level is assigned to every thread
  - Threads at the highest priority level run first
  - On a certain priority level: round-robin (one after the other)
  - Priority can be set dynamically and explicitly
- Optional preemption and time-slicing
  - Time-slicing is allowed but not mandatory – depends on the current implementation
  - The following scheduling points are defined anyway
    - wait
    - sleep
    - wait on a lock
    - wait on I/O
    - yield
    - stop
    - suspend

```
// Program tests whether or not the scheduler is preemtiv

class TestScheduler extends Thread {
        static final int N = 10000000;     // Duration
       /*static final int N = 10;              // Duration */

        String id;

        public TestScheduler (String s) { id = s; };

        public void doCalc() { // CPU-intensive calculation
                int i;
                for (i = 0; i < N; i++) {}
        }

        public void run() {
                int i;
                for (i = 0; i < 5; i++) {
                        doCalc();
                        System.out.println(id);
}       }       }


public class TestSched{

        public static void main(String args[]) {
                TestScheduler t1, t2, t3;
                t1 = new TestScheduler("Thread-1");
                t2 = new TestScheduler("Thread-2");
                t3 = new TestScheduler("Thread-3");
                t1.start();
                t2.start();
                t3.start();
}       }
```

N=10000000

Thread-1
Thread-2
Thread-1
Thread-3
Thread-2
Thread-1
Thread-3
Thread-2
Thread-1
Thread-3
Thread-2
Thread-1
Thread-3
Thread-2
Thread-3

N=10 or no-preemption

Thread-1
Thread-1
Thread-1
Thread-1
Thread-1
Thread-2
Thread-2
Thread-2
Thread-2
Thread-2
Thread-3
Thread-3
Thread-3
Thread-3
Thread-3

Problems of Priority Scheduling

- Most general approach – with “dangers”

- *Starvation danger*
  - Ready threads with a high priority let never run threads with lower priority

- *Priority inversion*
  - If a thread keeps a lock, other threads, requiring the same lock must wait – even with a higher priority
  - A thread with a low priority can thus slow down a number of high-priority threads

- Solution for priority inversion:
  - Thread T_1 with priority p_1 keeps lock L
  - Thread T_2 with priority $p_2 > p_1$ needs Lock L
  - Thread T_1 „inherits“ temporarily priority p_2, until the lock is released
  - After that T_1 has the priority p_1 again
  - Priority inheritance is implemented by many JAVA VMs – it is, however, not mandatory

- Priority in Java is defined in the range
  Thread.MIN_PRIORITY (1) ..
  Thread.MAX_NORM (5) .. (Default)
  Thread.MAX_PRIORITY (10)

Implementation of Threads

- *Green-thread (user-level) model*
  - Scheduling by the Java VM
  - Fast – no system calls are needed
  - Easier to port
  - I/O Operations must be *non-blocking*

- *Native-thread (system-level) model*
  - Java-Threads are mapped on operating system threads
  - Scheduling is done by the operating system
  - Threads become „heavier“
  - „Thread-pooling“ can be used
    - Instead of creating always new threads, threads are reused
    - Unused threads are stored in a pool
  - Usual implementation on Windows
  - Unix systems (Solaris, Linux) provide generally both models